6.9 STATE INFORMATION

In addition to user actions and feedback, UAN task descriptions must somehow specify when information about interface state must be retained and/or changed. This must be specified because interface state can affect interaction within a task. A simple example is seen in the earlier example of the task of selecting a file icon. Being selected has to do with the state of the file represented by the icon. Modes are another kind of interface state. (Despite what you may hear to the contrary, not all modes are bad.) So how is state information represented in the UAN? A new column is simply added to the task description.

At this point, the abstraction used for locality of definition says that interface state information should be kept separate from the way it might be presented in a display. The simplest example of this is seen in the now-familiar selection task. You may have noticed that all along, this has been called the `select file` task and not the `select file icon` task. The difference is subtle but important and can be completely missed if design is carried out using constructional thinking, such as which widgets are needed, rather than focusing on the behavioral domain.

It might seem as though a user is selecting the file icon, but the purpose of this task is for a user to choose a specific *file*, possibly as the object of a subsequent file operation, such as opening or deleting. The physical file itself is stored somewhere on a disk and cannot be made visible or touchable by the user. So in a graphical direct manipulation interface, the *interaction object*, in this case the file *icon*, serves as a visual surrogate for the file. The extent to which a user can naturally operate within this illusion (Weller & Hartson, 1992) with a feeling of directly manipulating the file, while actually indirectly manipulating it through its iconic representation in the interface, is a measure of the cognitive directness and effectiveness of the desktop metaphor. Thus, this difference should not be visible to the user, but it is still very important for the designer.

How does this affect the task description for the selection task? It simply means that the file and the file icon referring to it are kept separate in that description, remembering that *it is the file that is selected but the associated file icon that is highlighted.* Thus, highlighting in the interface represents state information about selection in the file system. Here's what the task description becomes when a third column is added to hold Interface State information:

TASK: **select file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'!	selected = file
M^		

Several parts of this task description must be defined further, probably in declarations. For example, `selected` is the name of a set that (in this case of single file selection) contains the name of the file that is currently selected. The user interaction designer must also explicitly declare the connection between the application entity (`file`) and the user interaction object (`file icon'`), in this general fashion:

```
file icon' ASSOCIATED WITH file
```

The final determination of how things such as selection are represented, though, should be based on what works best for the designer. The UAN is intended to be practical. Thus, the distinction between physical and logical selection should be maintained only as far as it is useful.

An example of a case where the distinction is, perhaps, not as useful to the designer is seen in the expression:

```
select (Open button)
```

Physical selection of the "Open" button causes logical selection (invocation) of the open command. In most direct manipulation interfaces, the task-oriented view does not include separate concepts for selecting a button and also for selecting the associated command. Even for the designer, the situation may be made simpler and/or clearer by referring only to the task or action of selecting the button. This same argument can, of course, be made for the case of the file icon and the file. If it suits the designer, the separate concept of a file icon can be omitted. Again, these fine distinctions are hidden from users but understood by designers.

6.10 CONDITIONS OF VIABILITY

A user interaction designer may not wish to deal with what it means to select something that is already selected. This difficulty can be avoided by allowing the `select file` task to apply only to files not already selected. This seems reasonable, but how can this be done in the UAN? We can qualify parts of the task description with a *condition of viability*. The `select file` task requires something that says,

```
for file icons not highlighted: ~[file icon] Mv^
```

The phrase before the colon is a condition of viability, and the expression after the colon is a conditional action. The general form for this kind of expression in the UAN is a condition, with a colon separating it from the action to which it applies:

```
condition: action
```

The scope of the condition is understood to include only the one action. Additionally, if the condition applies to a sequence of actions on the same line, it is written like this:

```
condition: action1 action2 action3
```

In such a statement, the scope is understood to include the actions in that one line. However, when the condition applies to actions on more than one line, parentheses are used to denote the scope in the following way:

```
condition:
  (action1
  action2
  action3)
```

A condition of viability acts as a precondition that must be satisfied—that is, have a true value—in order for user actions within its scope to be performed as part of the task being described. A condition of viability with a false value does not necessarily mean that a user cannot perform the corresponding actions; it just means that, even if they are performed, *the user is not performing this particular task.* The same actions, however, might be part of another task in the overall set of asynchronous tasks that comprise an interface.

When deciding how to use the UAN for a condition that requires the file icon to be unhighlighted, remember that for the file selection task, the following expression described the condition of being unhighlighted:

```
file icon-!
```

This is the same expression used for representing feedback earlier, where the expression served as an imperative; that is, it meant, "Make the file icon unhighlighted." Now we will use this expression as a condition of viability, where it says "file icon is unhighlighted," a logical proposition that can have a true or false value. The way you can tell the difference between the two cases is the context within a task description.

The first line of user actions of the `select file` task, with the new condition of viability, now becomes

```
file icon'-!: ~[file icon'] Mv
```

The colon indicates an if–then type of construct here, so you read this as follows:

"If the specific file icon of interest is not highlighted, then move the cursor to the context of that file icon and depress the mouse button."

As still more detail is added, the UAN task description now becomes

TASK: **select file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
file icon'-!: (~[file icon'] Mv	file icon'!	selected = file
M^)		

This example shows the use of parentheses in the User Action column to show that the whole task lies within the scope of the condition of viability. Just as before, the occurrence of `file icon'` in the condition of viability is bound to its other occurrences, meaning that they all refer to the same file icon. As a matter of practice, a shorter version of this expression may be easier to use. It amounts to a kind of built-in binding. Combining the conditional use of ! with the object of cursor movement, the following expression means, more directly, "move the cursor to a specific unhighlighted file icon and depress the mouse button":

```
~[file icon'-!] Mv
```

The task description, somewhat simpler than the previous one that describes exactly the same thing, becomes

TASK: **select file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'-!] Mv	file icon'!	selected = file
M^		

While this is a good way to represent the `select file` task, to be complete, the designer may also wish to specify what happens if the user does, in fact, try to select a file that is already selected. The most sensible result (and the one that happens on many computers) is that the file icon remains highlighted (and therefore the associated file remains selected) and nothing changes. How can this be included in the `select file` task description? The simplest way is to remove the condition of viability and define the meaning of ! as applied to file icons to be null (i.e., nothing changes in the icon appearance) if the file icon is already highlighted. The task description is correct in either case.

An alternative is to use a *condition of viability in the Interface Feedback column,* making the highlighting action dependent on the object's not being already highlighted. This covers both cases (i.e., selected icon already highlighted and not already highlighted). The line of feedback in the UAN description then becomes:

```
file icon'-!: file icon'!
```

This is read, "If a file icon is not highlighted, then highlight it" (in response to the `Mv` in the User Action column). Because it covers both cases, the following is probably a better task description than the previous ones:

TASK: **select file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file
M^		

6.11 EXTENSIBILITY OF THE UAN

To further extend your understanding of UAN, we will introduce a slightly different example: the task of moving a file icon on the desktop. Here is a prose description of that task:

1. Move the cursor to the file icon. Depress and hold down the mouse button. Depressing the mouse button selects the file, indicated by the highlighting of its icon.
2. With the button held down, move the cursor. An outline of the icon follows the cursor as you move it around.
3. Release the mouse button. The file icon is now moved to where you released the button, and the corresponding file remains selected.

Notice that, unlike the file selection task, this task changes the location of the file icon. Moving the file icon does not change the interface state relating to the file; it remains selected.

Exercise—UAN

GOAL: To produce a simple UAN task description.

MINIMUM TIME: About 10 minutes.

ACTIVITIES: Before looking at the development of the task description that follows, try to write a UAN task description for the `move file icon` task. You won't have all the notation you need, but this will give you a better appreciation for what is needed—and it won't take you much time. Start by drawing three columns—for the user actions, interface feedback, and interface state. Then go through the prose description line by line, trying to translate the prose words and phrases into UAN symbols. Also look back at the final UAN version of the `select file` task; you will find that much of it can be used in the UAN description of this new task.

For those parts of the prose you don't yet know how to represent in UAN, make up any symbols you might need to create a reasonably complete and precise description of this task. This will help you understand the power and the extensibility of the UAN.

DELIVERABLES: A UAN task description for the `move file icon` task.

Possible Exercise Solution

Here is the beginning of a task description for the `move file icon` task:

TASK: **move file icon**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file
~[x,y]* ~[x',y']	file icon' > ~	
M^		

It was easy to get started on this task description because it begins the same way that the `select file` task did. The first line of user actions and its associated feedback and state are, in fact, identical. The second line of user actions contains something new. The notation

```
~[x,y]
```

means that a user moves the cursor to some arbitrary point on the screen, *x, y*. The star, or asterisk (*), is the Kleene iterative closure star from regular languages and refers to the *repetition* of user actions. Literally, it means to do the preceding action zero or more times. When you add the star to the cursor movement, you get

```
~[x,y]*
```

which means to move the cursor arbitrarily around the screen going to zero or more points.

If you have a mind for precision, you might think that because the name *x, y* is used for the point coordinates for every repetition, binding would require that to be the same point every time. To avoid this difficulty, because it is impossible to use different coordinate names for each instance of the repetition in

```
~[x,y]*
```

it is necessary to assume that instances of *x, y* are not bound to each other (i.e., are not the same point on the screen). This convention allows the reference to a point *x, y* in the expression

```
~[x,y]*
```

to be a reference to an arbitrary number of points on the screen.

The expression

```
~[x',y']
```

that follows is also a move of the cursor to some point, this time a single specific point `x',y'`. This point is special in the task description because it is the final point in the cursor movement. The primes in the point coordinates are used to indicate reference to that specific point. The feedback for this cursor movement

```
file icon > ~
```

is an example of an object following the cursor (specifically, dragging the file icon) that was introduced in Section 6.8.

A closer look at this UAN task description shows that it isn't precisely what the prose description of the task said. Namely, Step 2 said that just the outline of the file icon follows the cursor. However, the UAN so far has not given us any way to refer to the outline of an icon. This is where the extensibility of the notation comes in. Remember, that is the topic of this section. So make something up! How about a function such as

```
outline(file icon')
```

This function is used in the UAN task description to say that just an outline of the file icon follows the cursor when it moves, but what happens when the user releases the mouse button? Then the complete file icon, not just its outline, appears on the spot where the user releases the mouse button. This sounds like the display function introduced at the end of Section 6.8. Thus, the feedback associated with releasing the mouse button could be represented in UAN as

```
@x',y' display(file icon')
```

Note that a binding is being used with the coordinate name `x',y'` from the User Actions column, to specify that the file icon is displayed at the point where the cursor stops moving. Putting these changes into the task description gives

TASK: **move file icon**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file
~[x,y]* ~[x',y']	outline(file icon') > ~	
M^	@x',y' display(file icon')	

There is still a small problem with precision here. While the file icon is displayed, in its final position (`@x',y'`), technically, it is still displayed in its original position, too. To avoid this, we can precede the display function with another one:

```
erase(file icon)
```

This making up of new functions or other notation, as needed, is fully within the spirit of extensibility of the UAN and its *open* symbology and structure. As long as we are creating new functions, let's introduce a redisplay function to combine both the erase and the display functions:

```
@x',y' redisplay(file icon)
```

Notice also, in the Interface State column, that this task leaves the file icon selected. Putting these changes into the task description gives

TASK: **move file icon**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file
~[x,y]* ~[x',y']	outline(file icon') > ~	
M^	@x',y' redisplay(file icon')	

Your possible solution may have been quite different from this one. However, if you captured this level of detail in your description, perhaps using different symbols or functions, your solution is probably a reasonable one. If you left out a lot of the detail, you might consider for a moment what would have happened in a real-world situation, where you, as the user interaction designer, handed your somewhat incomplete, imprecise UAN description to the user interface software designer and programmer. What would they have implemented? For how much of the design would they have had to guess what you meant? As stated earlier, when the interaction design is incomplete, the programmer sometimes ends up doing that design, often with unexpected or undesirable results. This is one of the situations that can be avoided by representing all details of the design using a technique such as the UAN.

6.12 TASK REPETITION

In the example of the `move file icon` task of the previous section, it was necessary to express arbitrary repetition of a user action or task. The UAN idiom

```
~[x,y]
```

denotes movement of the cursor to some (any) point on the screen. The expression

```
~[x,y]*
```

then refers to an arbitrary number (zero or more) of occurrences of this kind of cursor movement.

As a variation of this expression for task repetition, there is also the UAN expression

```
~[x,y]+
```

which means that cursor movement will occur one or more times; that is, at least once. Here the + symbol, like the * symbol, is taken from the language of regular expressions.

Sometimes, an interaction designer will wish to require a user to perform an action or task some specific number of times. If, for example, you wished to represent exactly three cursor movements, then (per regular expression notation), you would use a 3 as a superscript for the expression, as follows:

```
~[x,y]3
```

A special case of task repetition occurs when a task or action is *optional*—that is, a user may or may not perform it. As a regular expression, this might be written as

```
~[x,y](0 or 1)
```

literally meaning that the user performs the action either zero times or once. However, the UAN also distinguishes optionality as a special case and denotes it by enclosing the task or action in curly braces:

```
{~[x,y]}
```

6.13 MORE EXERCISES

To give you some more practice with UAN, let's try another task: Delete a file by dragging its icon on the desktop to the trash can icon. Details of this task vary, depending on your computer, so let's use the following prose description as the one for which you will write the UAN task description:

1. Move the cursor to the file icon representing the file to be deleted.
2. Depress the mouse button; the file icon highlights.
3. Keeping the mouse button depressed, move the cursor to drag an outline of the file icon over the trash icon. The trash icon highlights.
4. Release the mouse button; the file icon disappears, and the trash can does something else (e.g., bulges) to show that the file was removed.

Exercise—UAN

GOAL: To produce another UAN task description.

MINIMUM TIME: About 15 minutes.

ACTIVITIES: Now you have a little more experience with the UAN. Stretch your knowledge by trying to write a UAN task description for the task to `delete a file`, as described in the preceding prose. As with the previous exercise you tried, begin by drawing the three columns. Feel free to reuse any parts of the previous UAN descriptions as you write out this new task description.

DELIVERABLES: A UAN task description for the `delete a file` task.

Possible Exercise Solution

Because this starts out a lot like the `move file icon` task, use what you know from that task. That gets you this far into the task description:

TASK: **delete file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
`~[file icon'] Mv`	`file icon'-!: file icon'!`	`selected = file`
`~[x,y]* ~[x',y']`	`outline(file icon') > ~`	

This description still leaves the user stranded out in the middle of the screen at *x',y'*, holding down the mouse button. Instead of stopping at *x',y'*, the user needs to keep moving the file icon to the trash can icon, and then to release the mouse button there, deleting the file:

TASK: **delete file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
`~[file icon'] Mv`	`file icon'-!: file icon'!`	`selected = file`
`~[x,y]*`	`outline(file icon') > ~`	
`~[trash icon]`	`outline(file icon') > ~` `trash icon!`	

`M^`	`erase(file icon')`
	`erase(outline(file icon'))`
	`trash icon-!`
	`trash icon!!`
		`selected = null`

Note the second kind of highlighting for the trash icon, `trash icon!!`, which was used to indicate that the file was received in the trash can. Further note that in order to associate specific feedback—namely, highlighting of the trash icon when the cursor moves over it—the action

```
~[trash icon]
```

is written on a separate line from

```
~[x,y]*
```

This is in contrast to the `move file icon` task, which had no special feedback upon reaching `x',y'`. In that task, feedback occurred only upon release of the mouse button.

This leads to a realization about the expression

```
~[x,y]*
```

used to represent cursor movement. The ~ is used in the UAN to represent *path-independent cursor movement* of arbitrary distance. So the expression

```
~[object]
```

is a path-independent move to an object. Thus, the

```
~[x,y]*
```

is not really needed in front of a move to a specific location or object (icon), *unless* you want to use it as an action to associate with feedback for cursor movement, specifically to show the feedback for cursor movement at a given point within a task.

In the task descriptions for the `move file icon` task,

```
~[x,y]
```

and

```
~[x,y]*
```

were used as a user action to associate with the feedback

```
outline(file icon) > ~
```

to show feedback behavior during cursor movement. This separate user action is not needed for that purpose in the present task because it can be associated here with the path-independent movement to the trash icon. Here is the UAN description of the `delete file` task without that redundant step:

TASK: **delete file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file
~[trash icon]	outline(file icon') > ~ trash icon!	
M^	erase(file icon') erase(outline(file icon')) trash icon-! trash icon!!	selected = null

Up until this task, the UAN examples have been tasks that involve the interface only, without any connections to noninterface functionality. For these desktop examples, if there were some noninterface functionality, it would be in the computer's file system. The `select file` and `move file icon` tasks affected feedback and interface state but did not make any changes to the files themselves. (There is an assumption here that the system architecture is such that file selection state information resides in the interface, and the file system does not have *direct* access to such information.)

Although it is desirable, in reality this separation of user interface from noninterface functions often cannot be maintained and often simply cannot be done at all. In such cases, it is necessary to indicate connections that do exist. Thus, the present example, the task description for deleting a file, is the first one that has meaning (semantics) outside the user interface. Most user tasks do have semantic connections to noninterface functionality; after all, that's the real purpose of an interface—to give easy access to the functionality of the system. To serve this purpose, another column is needed—*Connection to Computation*—in the UAN task descriptions. This column is important to interaction designers because it is where they specify simple connections from the interface to the computational component. Software designers in the constructional domain might implement these connections, for example, as callbacks to functional routines from interface widgets.

TASK: **delete file**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
~[file icon'] Mv	file icon'-!: file icon'!	selected = file	
~[trash icon]	outline(file icon') > ~ trash icon!		
M^	erase(file icon') erase(outline(file icon')) trash icon-! trash icon!!	selected = null	mark file for deletion

Notice that in the new column, the file is to be marked for deletion, rather than to be deleted immediately. This is what actually happens on many desktop interfaces. At some later time, by some mysterious algorithm, the system decides to do some garbage collection and actually deletes all files marked for deletion. In the meantime, the file continues to reside in the trash can, which acts pretty much like a regular file folder.

In the next example, you are to delete several files at once from the desktop. This

task begins with selecting several files. Although there are many ways to do multiple file selection, each of which involves some complexity, only one way of performing this task is represented here. This method deletes files by using the *shift-select technique*, which involves holding the "Shift" key down while going around and clicking on file icons to be selected for deletion. When all the file icons the user wants to delete are selected, the user can then drag them as a group to the trash. Here is the prose description to be described in UAN:

1. Depress the "Shift" key, and hold it down.
2. Move the cursor to a file icon.
3. Click the mouse button; the file icon highlights.
4. Repeat the previous two steps as many times as desired.
5. Release the "Shift" key.
6. If desired, repeat the first five steps any number of times.
7. Move the cursor to any one of the selected file icons, depress the mouse button, and drag the selected group of file icons to the trash icon; the trash icon highlights.
8. Release the mouse button, dropping the file icons into the trash. The trash icon highlights (or bulges) again.

Exercise—UAN

GOAL: To produce a slightly more complicated UAN task description.

MINIMUM TIME: About 20 minutes.

ACTIVITIES: Even though this one is a bit more difficult, it still builds on what you have learned. Don't peek ahead to the possible solution! You need to be strengthening your UAN skills now. Make your best attempt at a UAN task description for the `delete multiple files` task just described in prose.

DELIVERABLES: A UAN task description for the `delete multiple files` task.

Possible Exercise Solution

This solution starts with just the part at the beginning, the first six steps in the prose description, which describe the multiple selection of files using the shift-select method:

TASK: **delete multiple files**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	CONNECTION TO COMPUTATION
(Sv (~[file icon'-!] Mv M^)+ S^)+	file icon'!	selected = selected U file	
rest of "delete multiple files" task...			

Here, S denotes the "Shift" key, where one of the keyboard keys (the "Shift" key) is used as a switchlike device (see Section 6.6.2) rather than as a character-string generator. Note the use of the + symbol (introduced in Section 6.12) in two places, to specify that particular actions are performed one or more times. In the preceding task description, this repetition applies to moving to the file icons and clicking on them while the "Shift" is key depressed. It also applies to the whole set of actions between, and including, depressing and releasing the "Shift" key. This means that a user can release the "Shift" key and still continue selection by depressing it again and clicking on more file icons.

Also note the built-in condition of viability, denoted by the -! in

```
[file icon'-!]
```

applied to the file icon in the user action on the second line. The icon cannot already be selected for a user to perform this action. If a user moves the cursor to a highlighted icon (in the second line) and depresses the mouse button, this task description simply does not indicate what would happen. Because the condition of viability is not met for this action, such a user action *does not apply to this task.* There may, however, be another task within the set of asynchronous tasks that comprise the interface (perhaps a `deselect` task), for which its conditions of viability are met and its description matches these user actions. That task description would say what results in this case.

As with many other formal specification techniques, the question of completeness is difficult. There is no way, in general, to determine whether all possible user behaviors are covered by the current set of task descriptions. This is an area of future work, drawing on software analysis techniques and tools that can process a set of task descriptions and identify problems with completeness, correctness, and consistency.

Returning to the present example, it is easy to make the case that a user might want to click on an already selected file icon during this task—perhaps due to a change of mind, deciding not to select it after all. As mentioned previously, the "Shift" key is used to allow multiple selection. The behavior of the "Shift" key when used for selection actually has slightly more meaning associated with it. In this case, holding down the "Shift" key puts the interface into a mode. (Remember, not all modes are bad; in fact, this one is pretty useful.) When the "Shift" key is depressed, a user can toggle between selection and deselection by successively clicking on the same file icon.

This provides an easy way to do multiple selection. Users can just hold down the "Shift" key and go around clicking on file icons. It also provides an easy way for users to change their minds about selecting some file; they can just click on its icon again, with the "Shift" key depressed, to deselect that file. This toggling behavior can easily be included in the task description. In the following UAN task description, note the interface feedback and interface state for the `Mv` action. Notice that the condition of functional viability on the file icon now moves from the User Actions column to the Interface Feedback column.

TASK: **delete multiple files**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
(Sv (~[file icon'] Mv M^)+ S^)+	file icon'-!: file icon'! file icon'!: file icon'-!	selected = selected U file selected = selected - file	
rest of "delete multiple files" task...			

To represent the rest of the `delete multiple files` task is just a matter of picking up any one of the selected file icons (denoted by `file icon!!`) and using

it to drag the whole group of selected file icons to the trash, as shown in the following:

TASK: **delete multiple files**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
(Sv (~[file icon'] Mv M^)+ S^)+	 file icon'-!: file icon'! file icon'!: file icon'-!	 selected = selected U file selected = selected - file	
~[file icon''!] Mv			
~[trash icon]	outline(∀file icon'!) > ~ trash icon!		
M^	erase(∀file icon'!) erase(outline(∀file icon'!)) trash icon!!	selected = null	mark selected files for deletion

We will return to this example of deleting multiple files in Chapter 7.

6.14 CONCLUSIONS ABOUT USE OF THE UAN

Having reached the end of this chapter, you probably have come to think of the UAN as very complex—all the details discussed in this chapter about moving the cursor and pushing buttons, highlighting, and so on. The level of detail you see in UAN descriptions may have surprised you. The alternative, however, is not to include so much detail in your designs and to leave the decisions about them up to a programmer at implementation time. The reasons why this is not a good idea should be abundantly clear to you by now.

Beyond this, however, the truth is that only a very small portion of the representation of a user interaction design in the UAN contains this kind of complex detail. These primitive user actions make up the *articulatory level* of the UAN, referring to the movement of a user's fingers in performing the physical actions. This level contains the low-level details of physical user actions (e.g., moving the cursor and clicking the mouse button) as a user interacts with devices. A complete

interaction description, however, is made up of many levels of abstraction built on top of the articulatory level, and these higher levels are made up of task names, not primitive user actions. These task macros and levels of abstraction are discussed in Chapter 7.

The complexity of the UAN as discussed in this present chapter is mostly hidden from readers and writers of the UAN, except when they are working at the articulatory level. Unless you are interested in exactly what physical actions are used to make selections, and so on, you need not see this level of complexity; instead you just see task names such as

```
select(object)
```

Above the articulatory level, interaction representations are in the form of a quasi-hierarchical task structure that looks very much like what you get from task analyses, except that the UAN also includes temporal relationships among tasks and subtasks, as discussed in Chapter 7. This next chapter also returns to the Calendar Management System and uses the UAN to represent its design.

REFERENCES

Card, S. K., & Moran, T. P. (1980). The Keystroke-Level Model for User Performance Time with Interactive Systems. *Communications of the ACM, 23*, 396–410.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Erlbaum.

Foley, J., Gibbs, C., Kim, W., & Kovacevic, S. (1988). A Knowledge-Based User Interface Management System. *Proceedings of CHI Conference on Human Factors in Computing Systems*, New York: ACM, 67–72.

Gould, J. D., & Lewis, C. (1985). Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM, 28*(3), 300–311.

Green, M. (1985). The University of Alberta User Interface Management System. *Computer Graphics, 19*(3), 205–213.

Grudin, J. (1991). Systematic Sources of Suboptimal Interface Design in Large Product Development Organizations. *Human-Computer Interaction, 6*(2).

Hartson, H. R., & Hix, D. (1989). Toward Empirically Derived Methodologies and Tools for Human-Computer Interface Development. *International Journal of Man-Machine Studies, 31*, 477–494.

Hartson, H. R., Siochi, A. C., & Hix, D. (1990). The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. *ACM Transactions on Information Systems, 8*(3), 181–203.

Jacob, R. J. K. (1986). A Specification Language for Direct Manipulation User Interfaces. *ACM Transactions on Graphics, 5*(4), 283–317.

Kieras, D. & Polson, P. G. (1985). An Approach to the Formal Analysis of User Complexity. *International Journal of Man-Machine Studies, 22,* 365–394.

Moran, T. P. (1981). The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems. *International Journal of Man-Machine Studies, 15,* 3–51.

Myers, B. A. (1988). *Creating User Interfaces by Demonstration.* Boston: Academic Press.

Payne, S. J. & Green, T. R. G. (1986). Task-Action Grammars: A Model of the Mental Representation of Task Languages. *Human-Computer Interaction, 2,* 93–133.

Reisner, P. (1981). Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Transactions on Software Engineering, SE-7,* 229–240.

Sharratt, B. (1990). Memory-Cognition-Action Tables: A Pragmatic Approach to Analytical Modelling. *Proceedings of Interact '90,* Amsterdam: Elsevier Science Publishers, 271–275.

Sibert, J. L., Hurley, W. D., & Bleser, T. W. (1988). Design and Implementation of an Object-Oriented User Interface Management System. In H. R. Hartson & D. Hix (Eds.), *Advances in Human-Computer Interaction,* Vol. 2 (pp. 175–213). Norwood, NJ: Ablex.

Wasserman, A. I. & Shewmake, D. T. (1985). The Role of Prototypes in the User Software Engineering Methodology. In H. R. Hartson (Ed.), *Advances in Human-Computer Interaction* (pp. 191–210). Norwood, NJ: Ablex.

Weller, H. G., & Hartson, H. R. (1992). Metaphors for the Nature of Human-Computer Interaction in an Empowering Environment: Interaction Style Influences the Manner of Human Accomplishment. *Computers in Human Behavior, 8,* 313–333.